UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS - ICEx
CIÊNCIA DA COMPUTAÇÃO

GABRIEL HENRIQUE SOUTO PIRES

DOCUMENTAÇÃO DO TRABALHO PRÁTICO
CONSTRUÇÃO DE UM ÍNDICE REMISSIVO (HASHING)

BELO HORIZONTE
2015

GABRIEL HENRIQUE SOUTO PIRES

DOCUMENTAÇÃO DO TRABALHO PRÁTICO

CONSTRUÇÃO DE UM ÍNDICE REMISSIVO (HASHING)

Trabalho prático apresentado à disciplina de Algoritmos e Estrutura de Dados 2 do curso de Ciência da Computação como requisito parcial para obtenção de nota.

BELO HORIZONTE

2015

1. Introdução

Um índice remissivo lista os termos e tópicos que são abordados num documento juntamente com páginas ou linhas em que aparecem. Esse tipo de índice é útil quando é preciso procurar por uma palavra específica em um documento muito longo.

Para criar tal índice, é possível criar uma estrutura de hashing para realizar as buscas. Esse tipo de estrutura é eficiente para esse tipo de problema pois faz com que a busca de um certo item no documento seja muito mais rápida, já que evitaríamos vasculhar boa parte do documento para encontrar o que queremos.

Nesse trabalho foi proposto que um programa em C fosse criado para construir um índice remissivo a partir de um arquivo lido em disco. Com o índice criado, foram escritas quatro funções. A primeira delas tem como objetivo salvar todas as palavras em ordem alfabética. A segunda função serve para achar uma palavra específica no documento e salvar em um arquivo de saída todas as linhas onde essa palavra aparece. A terceira função salva no arquivo de saída a palavra mais frequente no texto. Por fim, a quarta e última função imprime no arquivo de saída a palavra buscada e as linhas em que essa palavra aparece com mais frequência.

2. Implementação

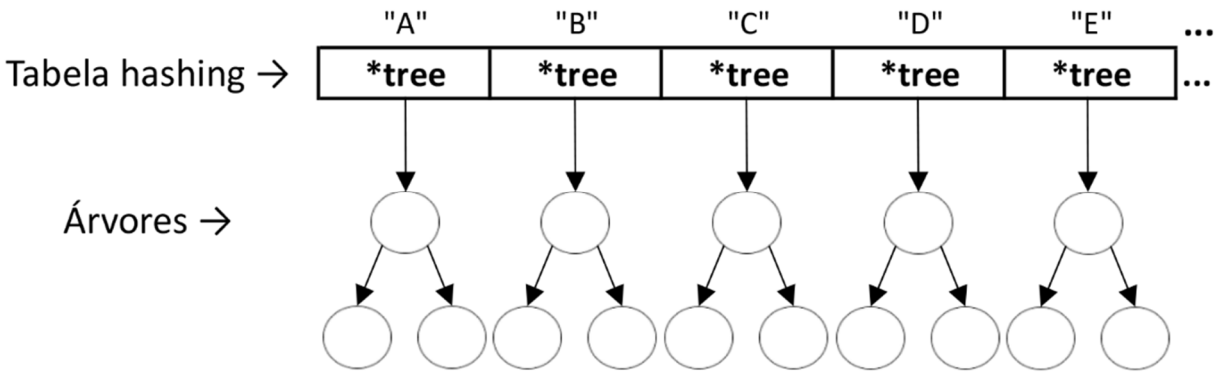
Para a implementação deste trabalho foi criado um tipo abstrato de dados do tipo Árvore Binária de Pesquisa, que armazena as palavras lidas do arquivo de entrada. A árvore é útil já que é necessário ordenar as palavras por ordem alfabética, e isso pode ser feito com uma simples função de caminhamento na árvore.

A estrutura de hashing foi implementada na forma de um vetor onde cada posição armazena um ponteiro para a raiz de uma árvore. Cada árvore armazena as palavras que começam com um caractere diferente do alfabeto ou número, supondo que os textos de entrada podem conter números. Assim o vetor de árvores contém 36 posições, 26 para as letras do alfabeto e 10 para os números de 0 a 9.

Se a palavra “Berlim” for inserida na tabela por exemplo, sabemos que ela começa com a letra “B”, então a palavra será inserida na árvore que armazena todas as palavras começadas com “B”, no caso a árvore contida na posição 11 da tabela, já que optei por reservar as 10 primeiras para os números.

Uma vez que cada palavra é inserida em uma árvore diferente de acordo com o primeiro caractere, o programa sabe exatamente onde procurar uma palavra que começa com “B” por exemplo, fazendo com que a busca por essa palavra seja muito mais rápida e eficiente.

A tabela de árvores pode ser representada com o diagrama abaixo.



2.1. Função principal

A função main recebe como parâmetro o número da parte a ser executada, o nome do arquivo de entrada, o nome do arquivo de saída, e no caso das partes 2 e 4 a main também recebe como parâmetro a palavra a ser buscada. Ainda na main são criados o vetor que guarda as árvores e o vetor de palavras que é usado na hora de inserir o conteúdo da entrada na árvore de palavras.

2.2. Função leArquivo

Para ler o arquivo de entrada é usada a função leArquivo que recebe como parâmetro a variável do tipo FILE do arquivo de entrada e o nome do arquivo de saída escolhido pelo usuário. Essa função é usada para preencher o vetor de palavras criado na main. Para preencher o vetor de palavras, a função conta quantas palavras existem no no arquivo de entrada e aloca um vetor com uma posição para cada uma delas. Cada posição do vetor contém uma struct do tipo Chave que tem 2 atributos, uma string “char *palavra” e um vetor de inteiros “int linhas[]”. Após alocar o vetor auxiliar com espaço suficiente para guardar todas as palavras da entrada, a função copia as palavras do arquivo de texto no vetor, guardando também o número da linha em cada uma aparece. Depois de ter todas as palavras no vetor, a função compara as palavras para remover as que repetem do vetor, e adicionar no vetor de linhas cada linha em que a palavra aparece. Ao final dessa função teremos um vetor com cada palavra do arquivo de entrada e com cada linha em que cada palavra aparece. Esse vetor é utilizado para inserir as palavras na árvore de pesquisa.

2.3. Função insere_elemento

Com todas as palavras da entrada salvas no vetor criado na parte 2.2, as palavras puderam ser inseridas na árvore de pesquisa por meio da função insere_elemento. Essa função testa se a árvore existe e faz a inserção da palavra na posição adequada de acordo com a ordem alfabética, se a palavra

a ser inserida na árvore é menor do que a raiz, ela é inserida à esquerda da raiz, se ela é maior então é inserida à direita. Não haverá o caso em que a palavra é igual à raiz, pois esse problema foi eliminado na função lerArquivo, uma vez que lá foram eliminadas as palavras repetidas do vetor.

2.4. Parte 1

Na parte 1 foi pedido que as palavras e os números das linhas onde elas aparecem fossem salvos em um arquivo em ordem alfabética. Para isso foi usada a função caminhar.

2.4.1. Função caminhar

A função caminhar é basicamente uma função de caminhamento em árvore binária. Foi utilizado o tipo de caminhamento central para retornar as palavras em ordem alfabética. Essa função é chamada para cada posição do vetor de árvores. Cada vez que a função é chamada, ela imprime todas as palavras presentes na árvore em um arquivo de saída.

2.5. Parte 2

Na parte 2 do trabalho era necessário mostrar todas as linhas onde uma palavra informada pelo usuário aparece. Para isso foi usada a função pesquisa.

2.5.1. Função pesquisa

Na função pesquisa foi utilizado um método de pesquisa binária onde o critério para a pesquisa é a ordem alfabética que é testada pela função strcmp. Ao se encontrar a palavra desejada, o seu vetor de linhas é impresso, ou seja, cada posição do atributo linhas[] da palavra que é do tipo Chave.

2.6. Parte 3

Na parte 3 a palavra mais frequente do texto teria de ser impressa no arquivo de saída, ou seja, a palavra que aparece mais vezes no texto. A função maisFrequente foi usada para isso.

2.6.1. Função maisFrequente

Nessa função, foi utilizado caminhamento em árvore novamente para achar a palavra que tem o maior vetor de linhas, isso significa que a palavra aparece mais no texto já que cada posição no vetor de linhas corresponde a uma aparição da palavra. A quantidade de aparições das palavras é armazenada em uma variável global, e caso a palavra visitada

atualmente na árvore apareça mais vezes do que o valor armazenado na variável global, essa palavra passa a ser a mais frequente.

Essa função recebe como parâmetro o ponteiro para a arvore de pesquisa, o arquivo de saída e o endereço de uma variável do tipo Chave que armazenará a palavra procurada. Depois de percorrer todas as árvores e achar a palavra mais frequente no texto, a variável do tipo Chave passada na função é alterada recebendo a Chave da palavra mais frequente e essa palavra é impressa na função main.

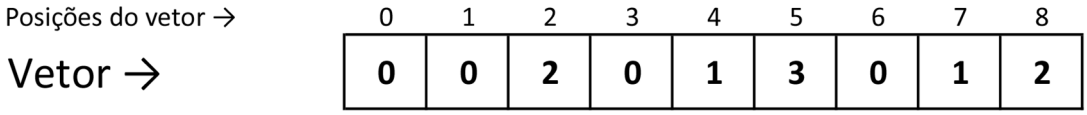
2.7. Parte 4

Na última parte do programa, uma palavra é informada e são encontradas as linhas onde essa palavra aparece com mais frequência. A função maisLinhas foi criada para resolver essa parte do problema.

2.7.1. Função maisLinhas

Esta função também usa uma forma de pesquisa binária para buscar a palavra informada. A diferença é que ao se encontrar a palavra, um vetor é criado com $n + 1$ posições para guardar a quantidade de aparições da palavra em cada linha, sendo n a última linha em que a palavra aparece. Depois de se criar esse vetor, são contadas quantas vezes a palavra apareceu em cada linha olhando o seu vetor de linhas posição a posição. Por exemplo, se a palavra aparece na linha 2 então se soma 1 à posição 2 desse meu vetor, isso é repetido até que todas as linhas sejam lidas. Um diagrama que representa esse vetor pode ser visto abaixo.

Linhas em que a palavra aparece (com repetição): 2, 2, 4, 5, 5, 5, 7, 8, 8



Após ter esse vetor preenchido, ele é percorrido e o maior valor é armazenado numa variável. Depois, são impressas todas as posições do vetor onde esse valor aparece, ou seja, são impressas todas as linhas onde a palavra aparece mais, que no caso do exemplo acima seria apenas a linha 5. Assim o problema da última parte do programa é resolvido.

3. Estudo de Complexidade

3.1. Função leArquivo

A função leArquivo executa alguns comandos O(1), depois entra em um loop $while$ para contar o número de palavras que executado n vezes, ou seja, O(n). Dentro desse loop são executados apenas comandos O(1). Depois são utilizados dois loops for aninhados para zerar o vetor de linhas, então $O(n^2)$.

Para ler o arquivo é usado um loop for dentro de um loop $while$ e cada loop executa operações O(1), então os dois loops juntos são $O(n^2)$.

Quando palavras iguais são achadas é necessário unir o vetor de linhas dessas palavras e remover as repetições do vetor. Isso é feito em um conjunto de loops onde são usados 3 loops for aninhados, então $O(n^3)$. Portanto temos: $O(n)+O(n^2)+O(n^2)+O(n^3)$ = **$O(n^3)$**.

3.2. Função Insere Elemento

A função insere_elemento percorre a árvore binária até achar o local de inserção. Como a função só executa comandos O(1) sua complexidade seria O(1), mas devido à recursividade a função se torna O(n). Sabemos também a altura máxima de uma árvore binária é log(n+1) então a complexidade no caso médio dessa função é O(log(n)). Portando consideramos que a complexidade é O(n).

3.3. Função caminhar

Essa função caminhar tem 2 loops for para imprimir a saída do programa, então O(n)+O(n), mas devido a sua recursividade, a função se torna O(n) caso na árvore a ser percorrida cada nó tenha apenas um filho, e O(log(n)) caso a árvore esteja balanceada. Mas como em todos os casos a função executará os loops para impressão da saída então a complexidade da função será O(n).

3.4. Função pesquisa

A função pesquisa também se comporta como a função caminhar, sendo O(n) a complexidade para qualquer caso.

3.5. Função maisFrequente

Essa função se comporta como uma pesquisa binária, sendo O(n) no pior caso e O(log(n)) no caso médio. Mas nessa função é usado um loop for então a complexidade dessa função é O(n).

3.6. Função maisLinhas

Essa função também tem comportamento similar à pesquisa binária, tendo também vários loops for separados cada um com complexidade O(n). então essa função também tem complexidade O(n).

3.7. Função main

A função main só executa comandos O(1) e tem loops for isolados, portanto temos que a complexidade dessa função é O(n).